□□□□□□□□

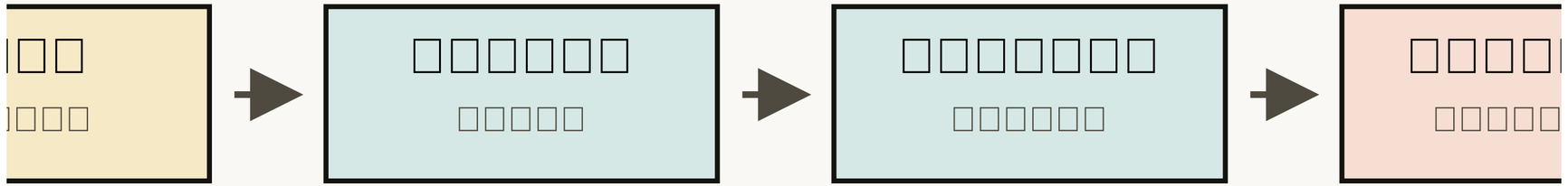
□2□

□□□□□

を読み、そのくせを集計

率で選び、文をつなげる

ループがあれば生成器は書ける



この考え方を使う最初の例

■■■■■■■■■■

単語に分ける出発点

の語の回数」を記録

ータで、操作を後から追加

●	今回の使い道
を分ける	単語に分ける出発点
	「単語 → 次の語の回数」を記録
ブル	「記号 → 実行する操作」を登録
	操作をあとから追加

□□□□□□□

だけでは、記号が単語に残留

、必要な文字だけを取り出す

字句解析と呼ぶ

```
 world! It's a nice day."

                          # □号が残る
a-z']+", text.lower())
orld', "it's", 'a', 'nice', 'day']
```

も、まずここから

回数だけでは、続きは不明

語を数えると、「この語の次」が見える

は、初めて出る語の回数も 0 から集計

```
ns import Counter, defaultdict

faultdict(Counter)

(words, words[1:]):
a][b] += 1
```

次の語の回数」の表

□□□□□□□□□□

図

の**状態**を保持

、次にすることを選択

態を更新

状態	表に入れるもの	ループで行うこと
今の単語	次に来た語と回数	次の語を選ぶ
数のスタック	記号と操作の関数	1トークンを実行する

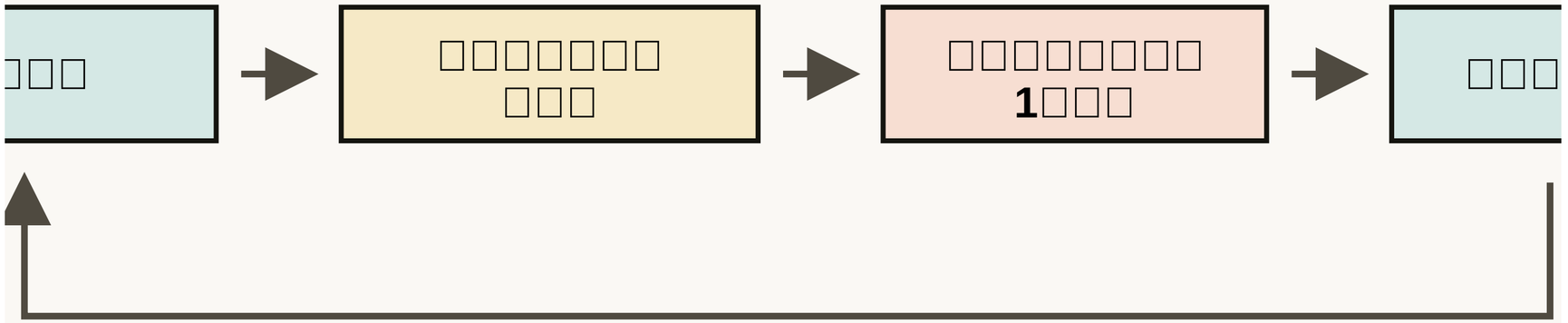
生成で使い、次に電卓でも利用

□□□□□

今の単語だけ

んだら、それが新しい状態

元の文章らしいくせは残留



□□□1□□□□□□□□□□

語を順に数え、辞書に加算

`"]` は「i の次に何が来たか」

like が 2/3、love が 1/3

```
e(tokens):
faultdict(Counter)
n zip(tokens, tokens[1:]):
a][b] += 1
le

 cats. I like dogs. I love cats."
table(text.lower().replace(".", "").split())
  -> Counter({'like': 2, 'love': 1})
] -> Counter({'cats': 1, 'dogs': 1})
```

□□□□□

`hoice` は等確率なので、ここでは不使用

`hoices` は重み付きで選択

ストなので、末尾の `[0]` が必要

```
counter):
st(counter.keys())
list(counter.values())
dom.choices(words, weights=weights)[0]

e["i"])    # 'like' □出やすく、'love' □たまに出る
```

、元の文章の頻度がそのまま残る

態、`table` が**表**

態を更新する**ループ**

に無ければ、そこで終了

```
able, start, n=30):
rt
word]
ange(n):
d not in table:
eak
 pick_next(table[word])
.append(word)
.join(result)
```

`T` 乗してから次の語を選択

いと最頻語に偏り、大きいとばらつく

とき、元の統計どおり

```
* (1 / temperature) for c in counter.values()]
```

重み `(3, 1)`	出力の傾向
`(59049, 1)`	最頻語に偏る。安定するが単調
`(3, 1)`	元の統計どおり
`(1.25, 1)`	ほぼ等確率。多様だが崩れやすい

ん。モデルの作り直しは不要

■■■■■■

見ると、出力は自然に

合わせは、見る語数とともに急増

、表に無い列ばかり

出力の傾向	問題
単語のつながりは自然	文の意味は崩れやすい
かなり自然	元の文章を写しやすい
ほぼ丸写し	組み合わせが表に無くなる

は、この表を学習した関数に置換

□□□□□□□□□□□□

□□□

□□□□□□

順位も不要

h、演算子なら pop して計算

ストが、そのままスタック

	やること	スタック
	数なので push	`[3]`
	数なので push	`[3, 4]`
	2つ pop して足し、結果を push	`[7]`

`7` 。先に pop されるのは `3`

□□□□□□□□□□

□覚えさせる

□の候補」だった表を「記号 → 関数」に置換

□デコレータを、そのまま `@word` として利用

しても、評価の骨格は不変

```
:
unc):
name] = func
 func
o
```

□□□□□□□□

]覚えさせる

分け、表にあれば関数を呼ぶ

として積む

しても、このループは不変

```
tack, text):
in text.split():
en in WORDS:
RDS[token](stack)

ack.append(int(token))
```

□□□□□

]覚えさせる

ループの3つがあれば十分

語を選び、電卓は関数を実行

プが、両方の骨格

	文章を作る機械	計算する機械
	今の単語	数のスタック
	次の語の回数	記号と操作
	確率に従って選ぶ	関数を実行する

□□□□□

:ファイル全体を1つの文字列に

イルは、行ごとに処理

`ncoding="utf-8"` を指定

```
a.txt", encoding="utf-8") as f:
ead()

a.txt", encoding="utf-8") as f:
n f:
 line.strip()
 line or line.startswith("#"):
ntinue
```

:よって文字化け

□□□□□□ `split`

`r"..."` で記述

は全部取り出し、 `sub` は置換

いと `match` は `None`

	意味
	任意の1文字
	数字 / 英数字と `_` / 空白
	いずれか / それ以外
	0回以上 / 1回以上 / 0か1回
	グループ。取り出したい部分

⬜⬜⬜

順に並べ、同じ回数なら辞書順

`WAP` は、スタックの上を操作

では、これらのワードを登録

```
items(), key=lambda kv: (-kv[1], kv[0]))
```

	動作	スタックの変化
	一番上を複製	(a → a a)
	一番上を捨てる	(a →)
	上2つを入れ替え	(a b → b a)
	2番目を上にコピー	(a b → a b a)

□□□□□□□□□

□□□□□□□□□□□

、語を状態にする機械が完成

同じ骨格のまま状態をスタックに置換

度や2語文脈で出力を比較

	ねらい
ークンに分ける	ファイル入出力、正規表現
る語」の表を作る	`defaultdict`、`Counter`、`zip`
て新しい一文を作る	重み付き抽選、生成ループ
操作に替え、電卓を動かす	スタック、ディスパッチ

☒☒☒☒☒

開く」から課題を開き、「ドライブにコピーを保存」

実行し、各課題の `TODO` を埋める

消さない

」のテストケースで採点するので消しても害はない

ードがあれば共有可